

高级语言程序设计

# 第 7 章 指针

华中师范大学物理学院 李安邦

# 第7章 指针

## 7.1 地址与指针

## 7.2 指针变量的定义和使用

## 7.3 指针与数组

## \*7.4 指针数组

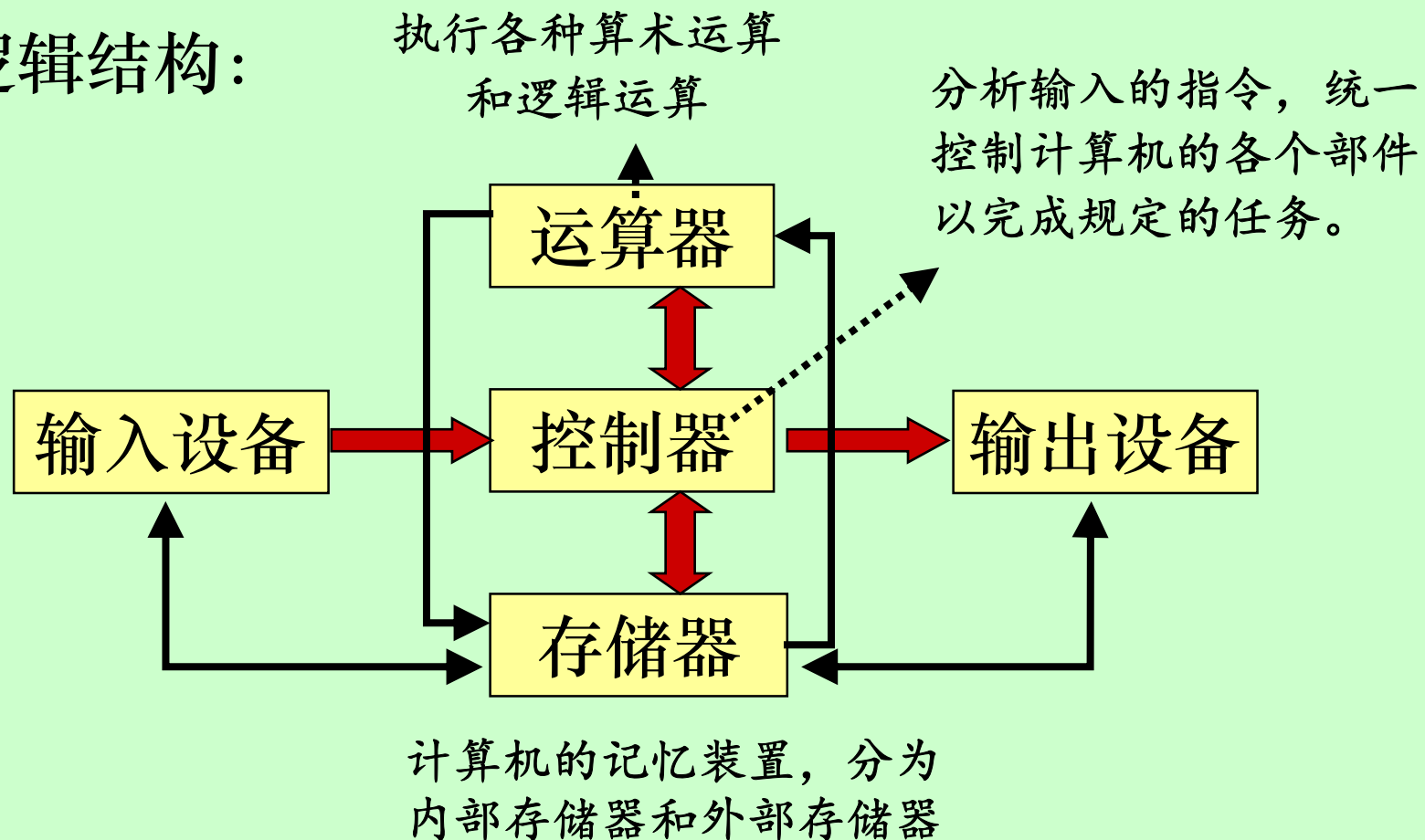
## 7.5 动态存储管理

## 7.6 指向函数的指针

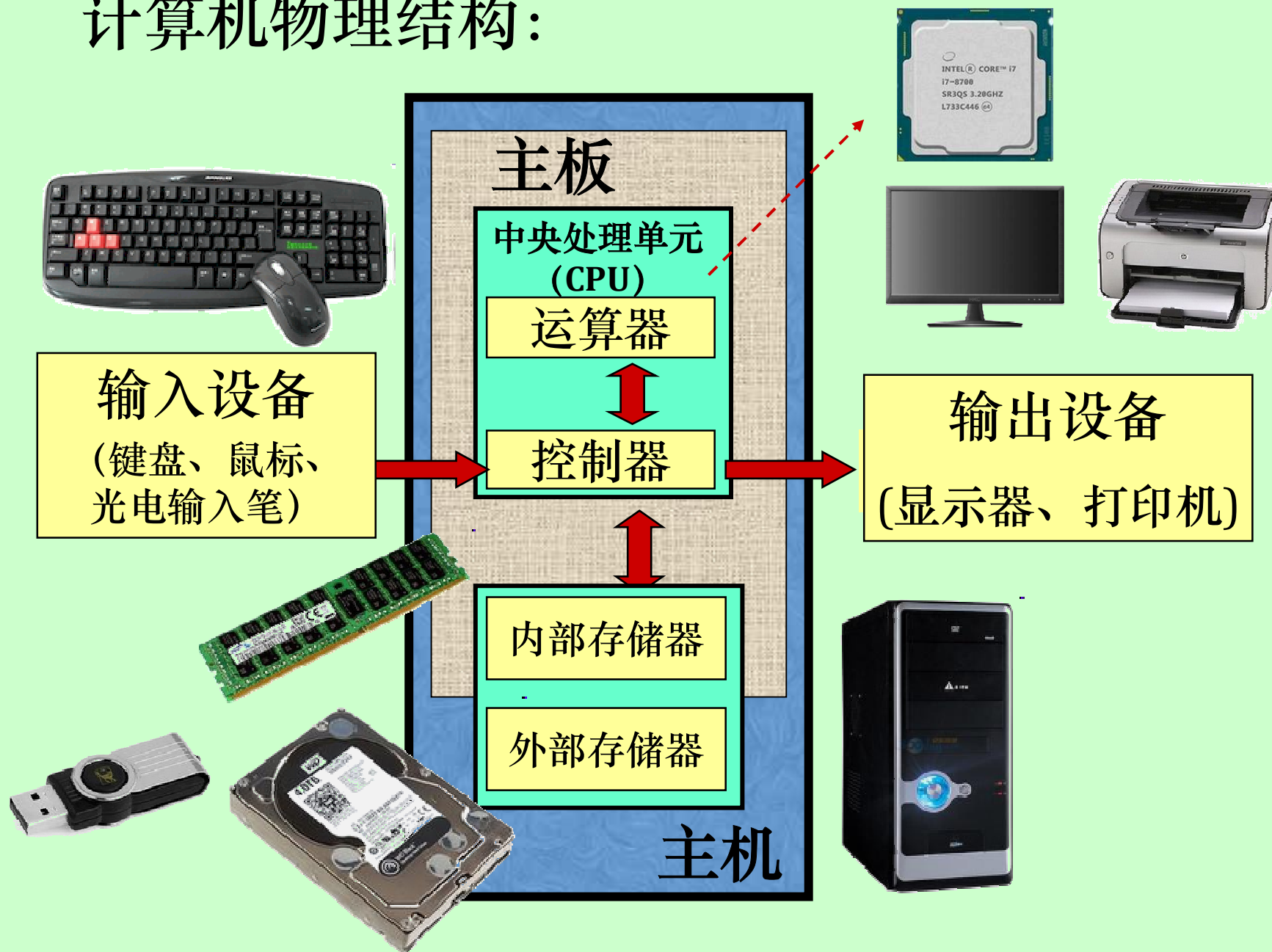
# 计算机硬件系统

计算机由**运算器**、**控制器**、**存储器**、**输入设备**和**输出设备**五个基本部分组成。它们通过**总线**连接。

逻辑结构：

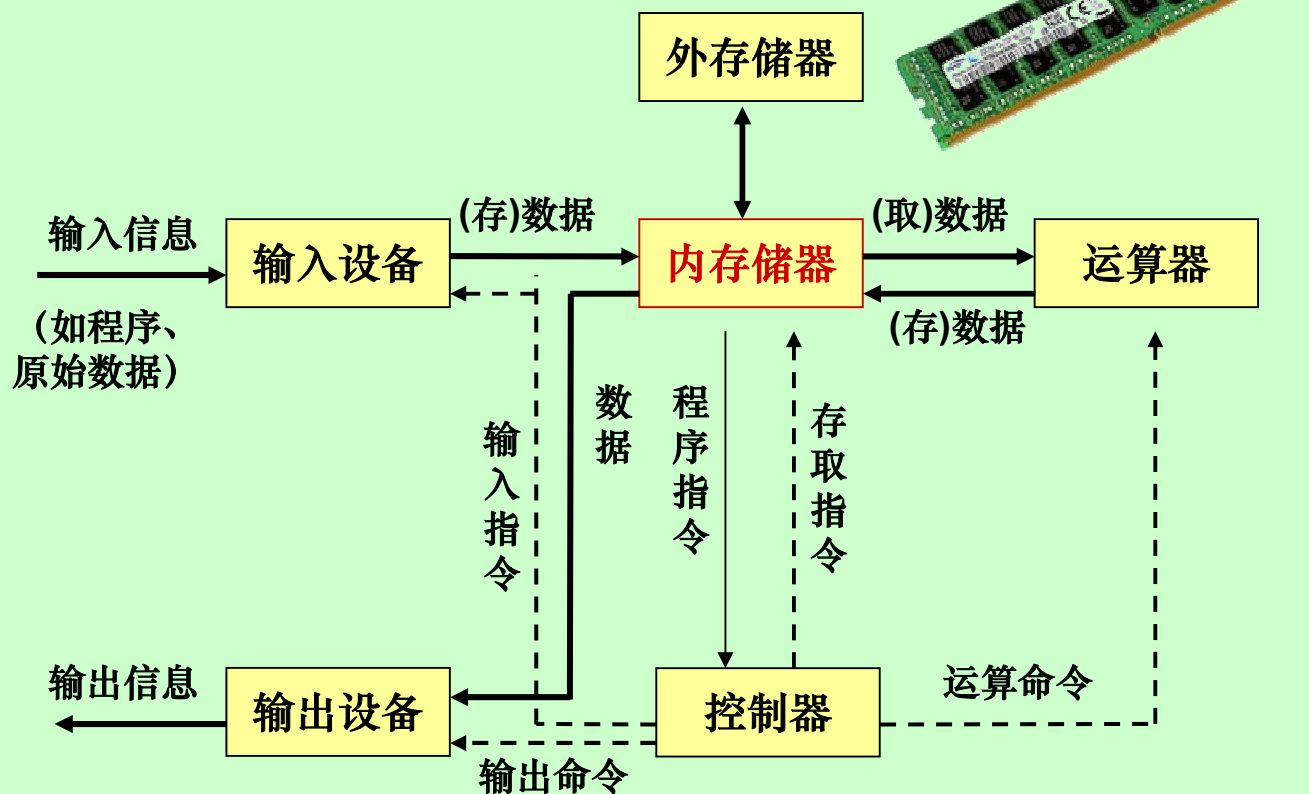


# 计算机物理结构:



# 内存储器处于数据信息流的中心

计算机硬件系统中的数据流：



实线箭头：数据信息；虚线箭头：控制信息。

# 地址与指针

- **程序执行中数据存于内存**。在可用期间数据有确定存储位置，占据一些存储单元。
- 内存每个单元（字节）都有一个编号：**地址**

## 2G内存，就是 $2^{20}$ 个字节。

给每个字节编号（地址）： $0 \sim 2^{20}-1$

[illegible]

- **机器语言通过地址访问数据。**高级语言用变量等作为存储单元/地址的抽象。
- **建立变量就是安排存储。**赋值时存入，用值时从中提取。
- **外部变量/静态变量有全局存在期，**程序执行前安排存储位置，保持到程序结束。
- **自动变量在函数调用时安排存储，**至函数结束。再调用时重新安排存储。

[illegible]

- 变量**存在期**就是它占据所安排存储的期间。
- 任何变量在存在期间总有确定存储位置，有固定**地址**。
- 变量存在时有地址，地址用二进制编码，因此可能成为程序处理的数据。

[illegible]



问题：地址作为数据有什么用？

- 若程序可以处理对象地址，就可通过地址处理相关对象。
- 对象（例如变量）地址也被作为数据，地址值/指针值。以地址为值的变量称为指针变量/指针 (pointer)。
- 指针是一种访问其他对象的手段，利用这种机制能更灵活方便地实施对各种对象的操作。

- 指针的主要操作

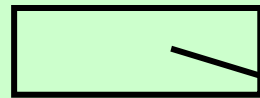
- 指针赋值：将程序对象的地址存入指针变量。

- 间接访问：通过指针访问被指对象。

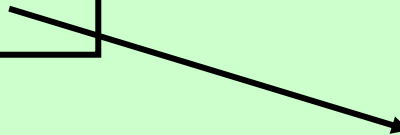
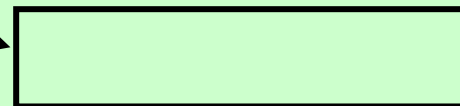
- 指针还能保存其他对象的地址。下面讨论以变量为例。

- 指针 p 保存着变量 x 的地址，也说指针 p 指向 变量 x：

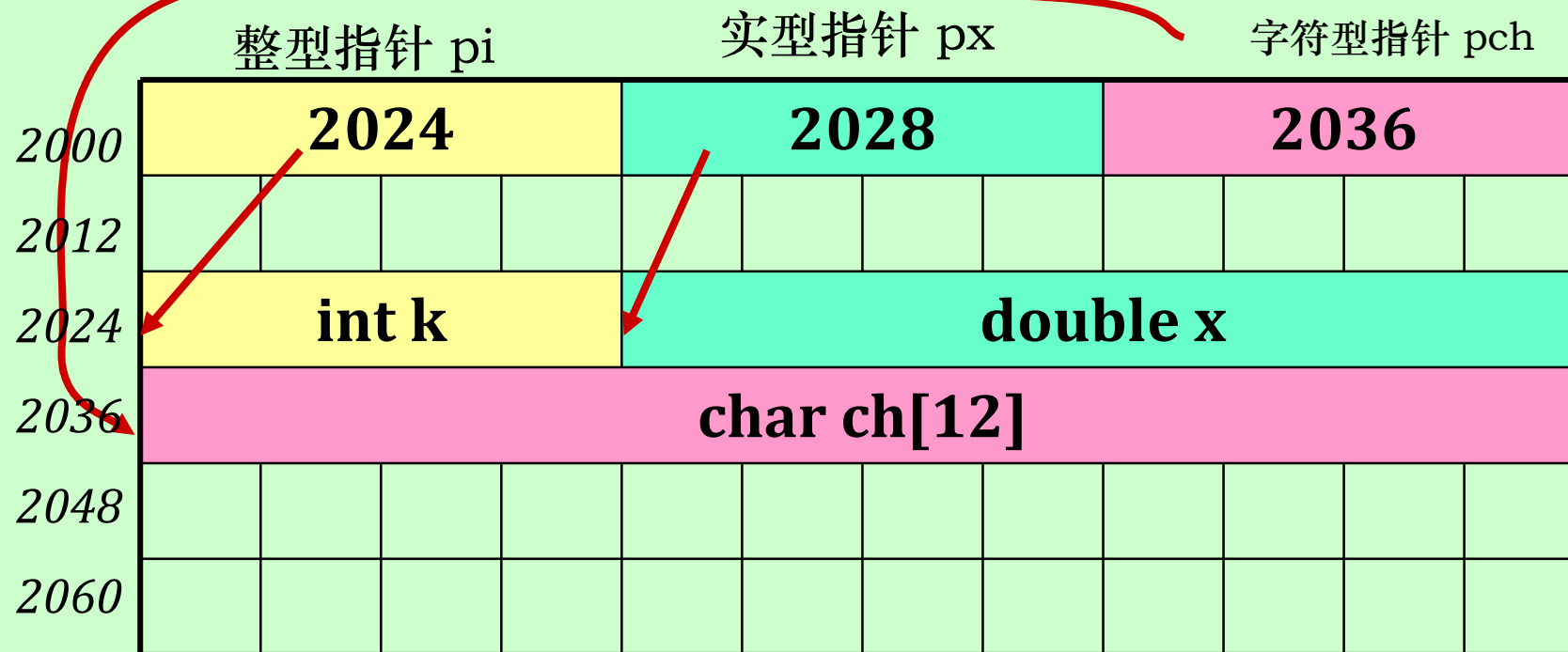
指针变量 p



变量 x



指针保存着变量的地址，也说指针 **指向** 变量：

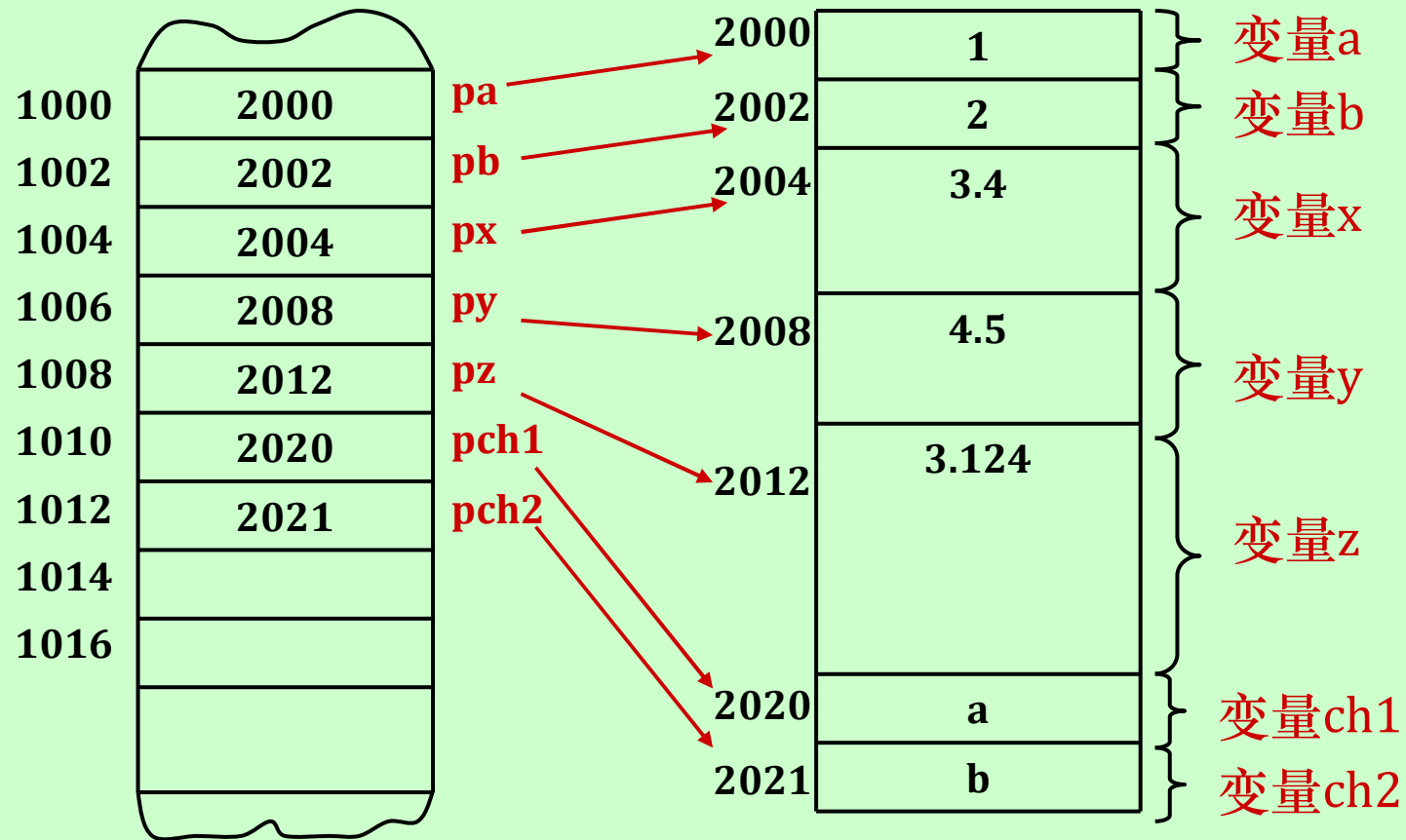


不必关心变量地址的具体数值，重要的是理解指向关系。

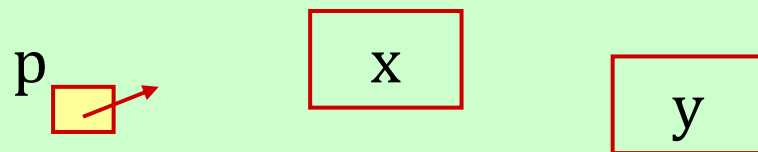
问：程序运行时指针变量在内存占多少位？与类型有关吗？

答：与类型无类，取决于编译器。经由32位编译器编译出来，都只占32位(4个字节)。经由64位编译器编译出来，都占64位(8个字节)。

## 指针 指向 变量（另一种常见的示意图）：



在这种示意图中，内存被画成竖向一维格子，指针变量和普通变量分开绘制。



- 指针是变量，可赋值，其指向可以改变。
- 现在 `p` 指向 `x`，以后可能指向 `y`。
- 通过 `p` 访问被指对象的语句目前访问 `x`，后来就访问 `y`。这种新的灵活性很有用。
- 在C/C++中使用指针常能写出更简洁有效的程序。有些问题必须用指针处理。
- 指针在大型复杂软件中使用广泛。指针使用的水平是评价C/C++程序设计能力的重要方面。
- C/C++指针灵活/功能强。掌握有难度，易用错，应特别注意。应特别注意使用指针的常见错误，**注意！**

# 第7章 指针

## 7.1 地址与指针

## 7.2 指针变量的定义和使用

### 7.2.1 指针变量的定义

### 7.2.2 指针操作

### 7.2.3 指针作为函数的参数

### 7.2.4 与指针有关的一些问题

## 7.3 指针与数组

## 7.4 指针数组

## 7.5 动态存储管理

## 7.6 指向函数的指针

## 7.2 指针变量的定义和使用

### 7.2.1 指针的定义

- 指针有类型，只能保存特定类型的变量的地址
- 指向 int 的指针 p 只能指向 int 变量。p 所指也看作 int。常说 int 指针 p 等。
- 定义指针需指明**指向类型**。（其它类型相似）
- 定义指针变量：

```
int *p, *q;
```

(理解：\*p 是int类型，或者说 p 是 int\* 类型)

- 指针变量可以与其他变量一起定义：

```
int *p, n, a[10], *q, *p1, m;
```

其它类型的指针变量的定义也是类似的。例如：

```
char *pch, *pch1;  //定义字符指针pch和pch1
```

```
double *px, *py;  //定义double 指针px和py
```

对每个基本数据类型都有相应的指针类型：int \*, double \*, char \*, bool \*, ..... C和/C++语言把指针类型也看作基本类型。

所有指针占用的存储都一样大，通常是一个机器字的大小。  
可以用运算符 sizeof 计算出来：

```
cout << "Size of int pointer : " << sizeof (p) << endl;
```

```
cout << "Size of char pointer : " << sizeof (pch) << endl;
```

```
cout << "Size of double pointer : " << sizeof (px) << endl;
```



## 7.2.2 指针操作

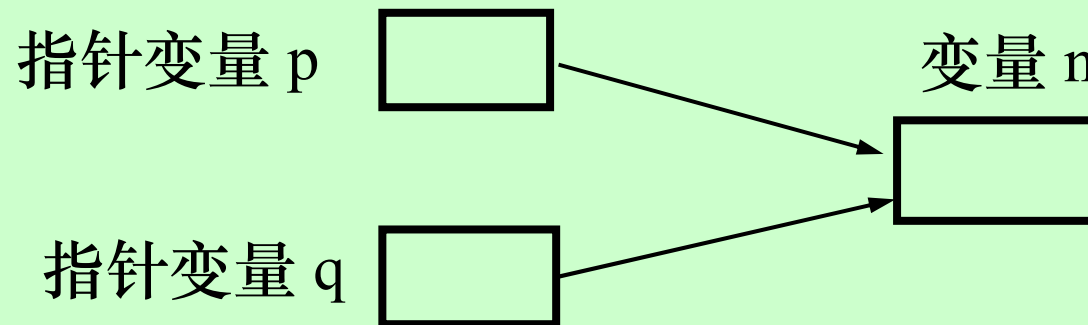
一元运算符：取地址运算符 **&** 和间接访问操作 **\***。

### 取地址运算

- **&** 写在变量描述（如变量名）前**取**变量地址，是对应类型的指针值，可赋给类型合适的指针。例：

`p = &n; q = p;      p1 = &a[1];`

- 多个指针可能同时指向同一变量。变量相等是值相等，两个指针变量相等说明它们指向程序里同一东西。



在定义指针变量时可以用合法的指针值初始化。

```
int n, *p = &n;
```

```
double x, y, *px = &x, *py = &y;
```

注意：

指针类型与变量类型如果不匹配，就不能赋值：

```
p = &x; px = &n; //不允许！
```

原因：

x 是 double 类型，&x 是 **double \*** 类型，不能赋给 **int \*** 类型的变量 p。（不会有自动类型转换）

**空指针值：**一个特殊指针值，表示指针变量闲置（未指向任何变量）。

唯一对任何指针类型都合法的值。

空指针值用0表示，标准库定义了符号常量 **NULL**：

**p = NULL;** 和 **p = 0;** 相同

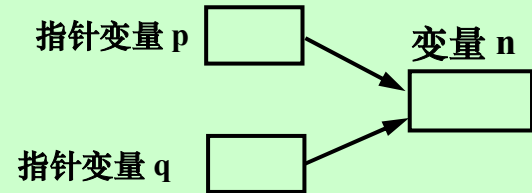
前一写法易看到是指针。

如果定义指针变量时没做初始化，外部变量和局部静态变量将自动初始化为空指针（0值），局部自动变量不自动初始化，建立后的值不确定。

## 间接访问

- 把间接访问操作符 `*` 写在指针前面，得到被指针所指的变量，可以像普通变量一样使用。
- 设 `p` 和 `q` 指向 `n`，间接赋值：

`*p = 17;`



这里写 `*p` 相当于直接写 `n`。另一个赋值：

`m = *p + *q * n;` // 访问`n`三次

`++ *p;` // 使变量`n`的值加1，变成18

`(*p)++;` // 使变量`n`的值再加1，变成19

`*p += *q + n;` // 变量`n`被赋以新值57

问：上文在定义指针并初始化时写有示例语句

“`int n = 10, *p = &n, *q = &n;`”，里面的“`*p = &n`”和“`*q = &n`”如何理解？

该语句中同时定义了 `int` 类型变量 `n` 和指向 `int` 类型的指针变量 `p` 与 `q`，并且用 `n` 的地址初始化了这两个指针变量。

它的实际含义相当于如下四条语句：

```
int n = 10; int *p, *q; p = &n; q = &n;
```

或者下面两个定义语句：

```
int n = 10; int *p = &n, *q = &n;
```

可见，混和定义形式不太好，拆分之后更易理解。

## 7.2.3 指针作为函数参数

- 指针作为函数参数有特殊意义，利用这种参数可写出能修改函数调用处的环境（在调用函数的位置能访问的变量全体）的函数。
- 前面函数的参数机制：
  - 在使用值参数时，函数中对形参的更改不会反映到实参。
  - 在使用数组作为参数时，可以修改数组元素的值，为什么？
  - C++ 中的引用型参数可以改变函数中的引用型参数的值。（但不能在 C 语言中使用）

要想让函数内部能改变调用处的变量（如局部变量），必须在函数里掌握（Hold）这个变量。

C++ 中的引用型参数能实现这一功能，

利用指针机制也可以实现调用处的变量这一功能：  
在调用时把变量的地址（地址值）通过指针参数传进函数，在函数内部对参数指针进行间接访问，就能完成对调用处的变量的各种操作（包括赋值）。

通过参数改变调用环境的方案包括三方面：

- 函数定义中用指针参数；
- 函数内部通过指针间接访问被指变量；
- 调用时以被操作变量的地址作为实参。 ➡

假设有某函数func的原型说明：

```
int func(int *p, int *q); //函数定义时用指针参数；
```

调用该函数的语句如下：

```
func (&m, &n); //函数调用时把变量地址传给函数。
```

则执行上面语句时的现场情况：

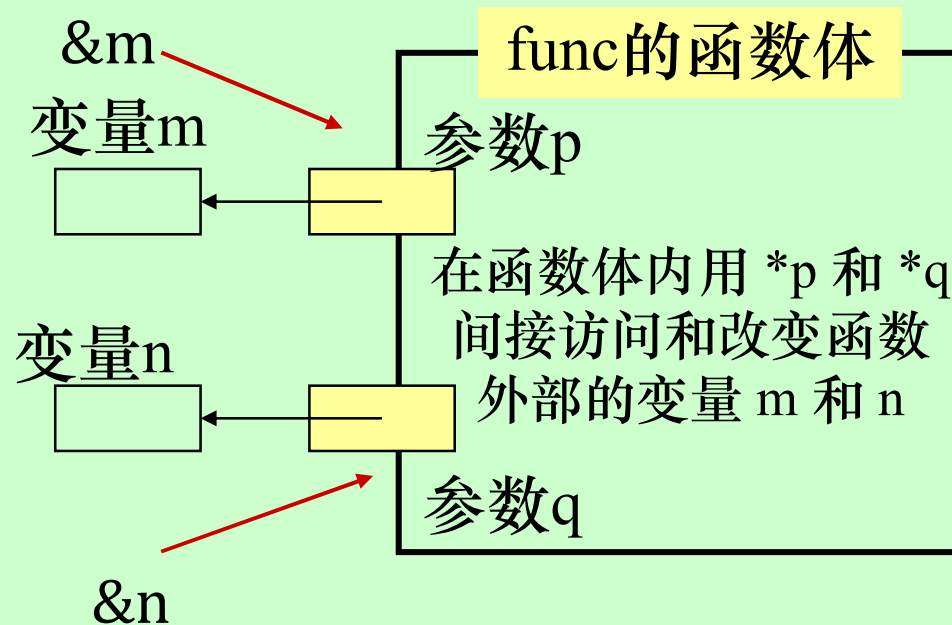
### 函数 func 的调用环境

函数定义：

```
int func(int *p, int *q) {  
    ...  
}
```

函数调用：

```
func(&m, &n);
```





【例7-1】使用指针作为函数参数，编写函数 `swapptr`，实现交换调用处的两个实参的值。

```
void swapptr(int *p, int *q) {  
    int t = *p;  
    *p = *q;  
    *q = t;  
}
```

两个参数的类型都是(`int *`)，调用时的实参必须是合法的整型变量地址。假设已有变量定义：`int a[10], k;`

则下面两个调用都是合法的：

```
swapptr(&a[0], &a[5]);  
swapptr(&a[1], &k);
```

表7-1 值参数、引用参数和指针参数的异同

	函数定义	函数调用	调用效果
值参数	int swap(int a, int b)	swap(m, n)	不改变实参的值
引用参数	int swapref(int &a, int &b) (&字符表示引用)	swapref(m, n)	改变实参的值
指针参数	int swapptr(int *p, int *q)	swapptr(&m, &n) (&字符表示取地址)	改变指针实参所指变量的值

函数可以有常参数。这种参数同样由实参提供初值，但在函数体里不允许对它们重新赋值。非指针参数（也就是传值参数）本来就不会被修改原始值，所以常参数只用于指针参数。

常参数的定义形式也是在函数参数描述中加const关键字，有两种用法：

(1) const关键词写在变量类型之前：

**const 类型 \*变量**

这种用法将限制修改指针指向的值。

(2) const关键词写在变量类型与变量名之间：

**类型 \*const 变量**

这种用法将限制指针的指向。

## 7.2.4 指针作为函数的返回值

函数的返回值也可以是指针（地址），这样的函数称为**返回指针的函数**，定义时把函数的返回值类型描述为相应的指针类型。

【例7-2】写一个函数，求出三个整数参数中的数值最大那个参数的地址值。

```
int * pmax3(int *pa, int *pb, int *pc) {
```

```
    int *p = pa; //定义局部指针变量并赋值为形参 pa所指向  
    的变量地址值
```

```
    if (*p < *pb)    p = pb;  
    if (*p < *pc)    p = pc;  
    return p;  
}
```

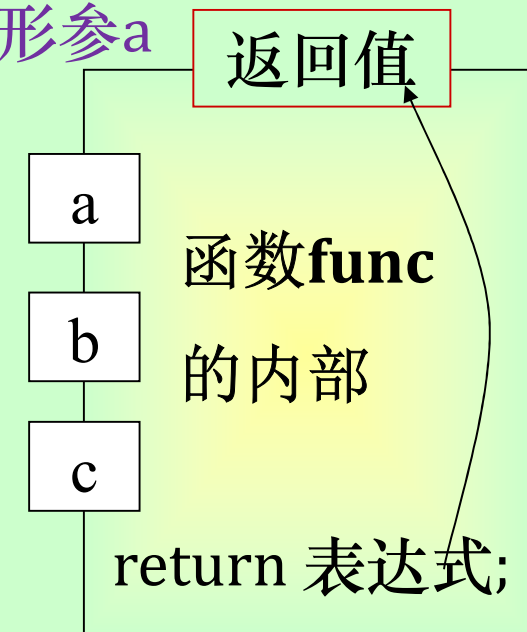
```
int main() {  
    int a = 5, b = 10, c = 3;  
    //调用 pmax3, 返回值赋给局部变量pm  
    int *pm = pmax3(&a, &b, &c);  
    cout << "Max= " << *pm << endl;  
    return 0;  
}
```

用指针作为函数返回值时需要特别注意：

函数运行结束时将销毁**内部定义的局部对象**（包括局部变量和形参），**函数返回的指针绝不能指向这些对象**。

系统不保证这些对象会一直有效，使用被销毁的对象可能会引发运行时错误，后果无法预料。

```
int * pmax2err(int a, int b, int c) {  
    int *p = &a; //定义局部指针变量并指向形参a  
    if (*p < b) p = &b;  
    if (*p < c) p = &c  
    return p; //是其中一个形参的地址  
}
```



## 7.2.5 与指针有关的一些问题

### 一、指针使用中的常见错误

#### (1) 对悬空指针进行间接访问

使用指针变量时，最重要的问题就是保证在对指针做间接访问（取值或赋值）之时，相应的指针已经指向了合法的变量。

当一个指针并没有保存当时合法的变量地址时，人们称它是**悬空指针**或者**野指针**。

使用指针时最常见的错误就是对**悬空指针进行间接访问**，即在一个指针并没有指向合法变量的情况下对它做间接访问。

常见错误写法：

```
int *p, n = 3; // p是悬空指针  
cout << *p; //错误：对悬空指针取值  
*p = 2; //错误：对悬空指针赋值
```

编译程序不能发现这类错误。

在运行时，间接访问悬空指针是严重错误，后果可能很严重。

需要编程者在使用指针时格外小心。

## (2) 间接访问空指针也同样是非法的

```
int *p= NULL;  
cout << "*p = " << *p << endl; //错误：对空指针取值  
*p = 10; //错误：对空指针赋值
```

编译时不报错，运行时报错（比悬空指针稍微安全一些）。

建议：在定义指针时总是把它初始化为空指针。

### (3) 对指针参数提供非指针值或悬空指针

在调用包含指针参数的函数时，与之对应的实参必须指向合法变量。常见错误：**把非指针值（常数或普通变量）或悬空指针作为函数的实参。**

例如在调用 `swapptr` 时，应该提供已指向合法变量的实参。

`swapptr(3, 5);` //错误：以常数值作为地址值提供给函数做实参

`swapptr(m, n);` //错误：以变量m和n的数值作为实参

由于类型不对，编译器能发现这些错误。

```
int *p1, n = 5;
```

```
swapptr(p1, &n);
```

 //悬空指针 p1 作为参数。运行错误!

格式化输入函数 `scanf`，对于待输入数值的变量前面一定要写 `&` 字符。



## 二、通用指针

类型(**void\***)，可以指向任何变量。

```
void *gp1,*gp2;
```

任何指针值可以赋给通用指针（不必转换）。例：

```
int n, *p;
```

```
double x, *q;
```

```
gp1 = &n;    // gp1指向n（值是n的地址）
```

```
gp2 = &x;    // gp2指向x
```

通用指针只取得了地址信息，并没有保存类型信息。

所以**不能对通用指针进行做间接运算**以取得被指对象的值。

通用指针唯一用途是作为中介，保存和提供指针值。

当一个通用指针已经通过赋值从一个特定类型的指针获得一个合法的地址值之后，可以把该通用指针的值赋给另一个同样类型普通的指针，而且在赋值时要进行强制类型转换：

```
p = (int *)gp1;
```

```
q = (double *) gp2;
```

下面赋值不合法：

```
q = (double *) gp1;    //wrong!
```

在学习 C 语言的动态分配存储时，将会使用通用指针。

但是学 C++ 的用户，在后面不需要使用通用指针。:)

# 第7章 指针

7.1 地址与指针

7.2 指针变量的定义和使用

7.3 指针与数组

7.3.1 指向数组元素的指针

7.3.2 数组写法与指针写法

7.3.3 基于指针运算的数组程序设计

7.3.4 数组参数与指针

\*7.3.5 多维数组作为参数的通用函数

7.3.6 指针与数组操作的程序实例

7.3.7 字符指针与字符数组

7.4 指针数组

7.5 动态存储管理

7.6 指向函数的指针

## 7.3 指针与数组

C 指针与数组关系密切，以指针为媒介可以完成各种数组操作。常能使程序更加简洁有效。

用指针做数组操作同样要**特别注意**越界错误。

指针和数组的关系是C语言特有的，除了由C 派生出的语言（如C++），一般语言中没有这种关系。

## 回顾数组的知识

数组（array）是多个同类型数据对象的组合。

定义数组变量（**定义数组**）时需说明：

- ◆ 数组元素类型，数组名
- ◆ 数组（变量）的元素个数（数组大小或长度）。  
用方括号内整型表达式说明元素个数，表达式应能静态确定值，可用**字面量**或**枚举常量**。

例： `int a[10]; double a1[100];`

`enum {NUM = 200}; char str[NUM];`

可定义**外部数组**和**局部数组**，包括**局部静态数组**（用static）。作用域与存在期与简单变量相同。

## 回顾数组的知识(续)

数组元素按顺序编号，首元素序号 0，其余顺序编号。**n元数组的元素编号是 0 到 n-1。**

例如定义：int a[10];

元素编号为 0、1、2、...、9。称为下标或指标。

**元素访问**通过[]运算符，优先级最高，运算对象是数组名和括号里表示下标的表达式。

表达式、语句里的“**[]**”写法称为**下标表达式**。

例：有上面定义后，可写：

a[0] = 1; a[1] = 1;

a[2] = a[0] + a[1];

a[3] = a[1] + a[2];

## 7.3.1 指向数组元素的指针

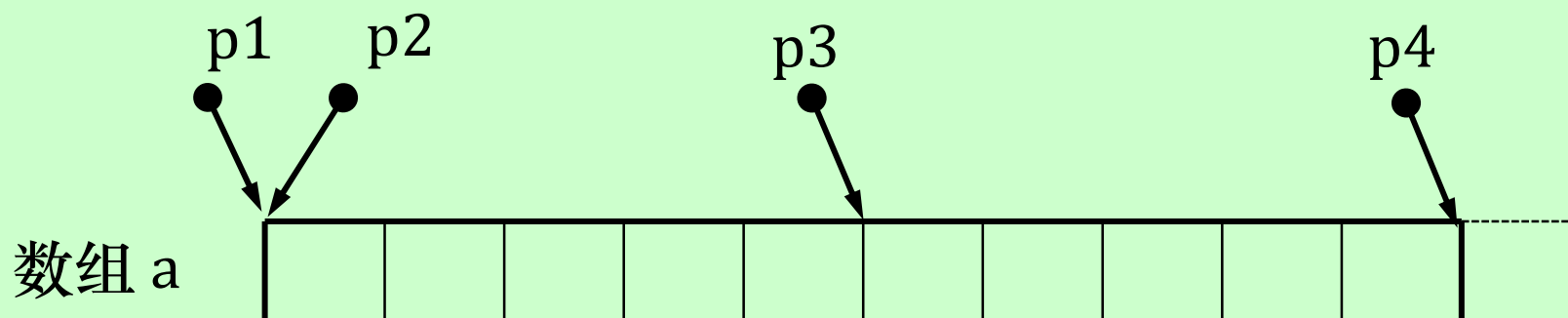
类型合适的指针可以指向数组元素。假定有定义：

```
int *p1, *p2, *p3, *p4;
```

```
int a[10] = {1,2,3,4,5,6,7,8,9,10};
```

则可以写：（直接写数组名得到数组首元素地址）

```
p1 = &a[0]; p2 = a; p3 = &a[5]; p4 = &a[10];
```



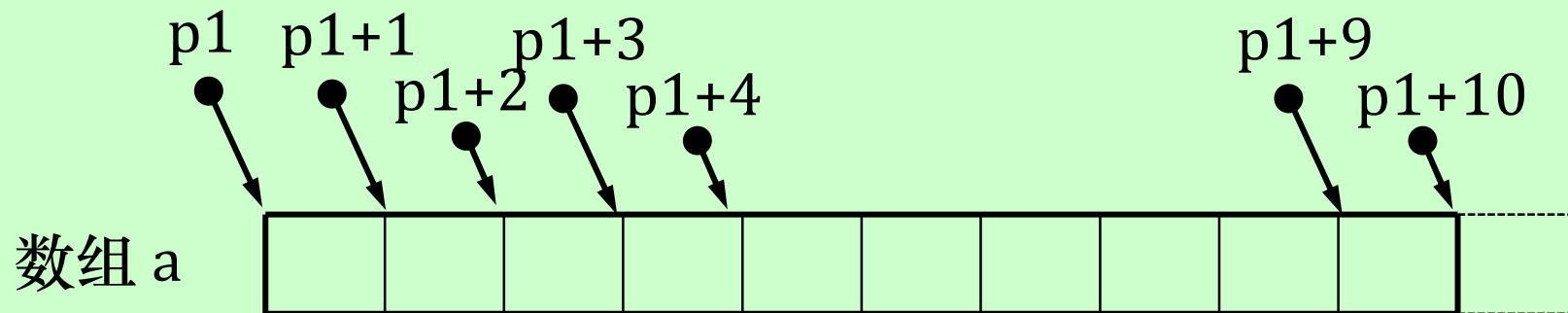
p4 没指向 a 的元素，是指向 a 最后元素向后一个位置。

C/C++ 语言保证这个地址存在，但写 **\*p4** 是错误的。

当指针  $p$  指向数组元素时说  $p$  **指到了数组里**。

这时由  $p$  可以访问被  $p$  指的元素（ $*p$  等价于访问被指元素），还可访问**数组的其他元素**。

1、可以对指针**加上一个整数值**，所得表达式也是一个合法指针。（该地址是向后移若干个元素的空间）



例：  $p1$  指向  $a$  首元素，值合法（ $a[0]$ 的地址）， $p1+1$ 也合法（ $a[1]$ 的地址）。 $p1+2$ 、 $p1+3$ 、...也合法，分别为  $a$  其他元素的地址。

由这些指针值可以间接访问  $a$  各元素。

例：  
 $*(p1 + 2) = 3;$  // 给 $a[2]$ 赋值  
 $p2 = p1 + 5;$  // 使 $p2$ 指向 $a[5]$

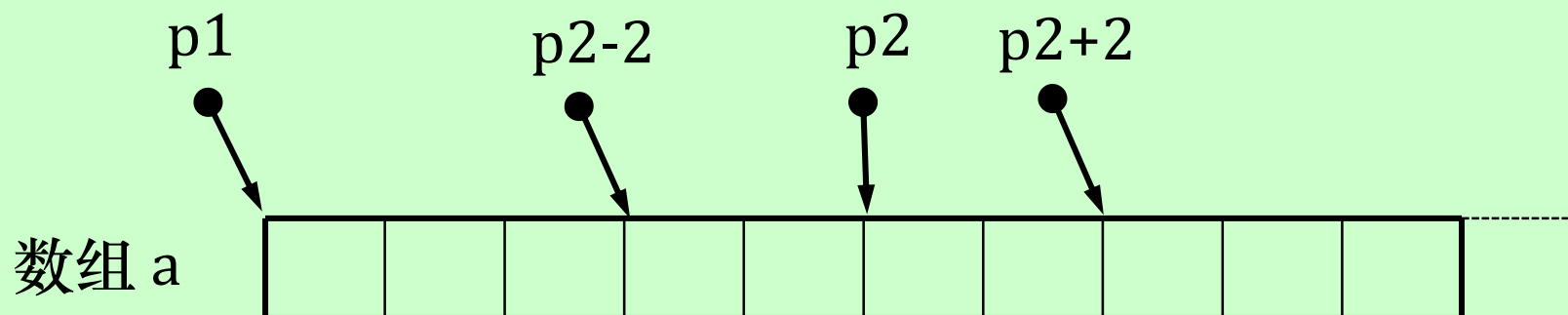


2、当指针**指向数组中间的元素**时，同样可用加法访问该位置之后的其他元素：

$*(p2 + 2) = 5;$      // 给a[7]赋值

还可**用减法**访问所指位置之前的元素：

$*(p2 - 2) = 4;$  /\* 给a[3]赋值 \*/



这类**由指针值出发进行的运算**称为**指针运算**。

通过指针访问数组元素时**必须保证不越界**。

运算取得的指针值（即使不间接访问）必须在数组范围内（可过末元素一位置），否则无定义。

3、可以对指针加减整数，进行指针更新：

用指针运算得到的值做指针更新：

```
p2 = p2 - 2; /* 这使p2改指向a[3] */
```

用增/减量操作做指针更新：

```
p3 = p2; ++p3;
```

```
--p2; p3 += 2;
```

通过指向数组的指针访问数组元素，必须保证所有间接访问都在数组合法范围之内。

不指在同一数组里时，求差和比较大小没有意义。

4、如果两个指针指在同一个数组里，可以求差，得到它们间的数组元素个数（带符号整数）。

$n = p3 - p2; //$  也可以求  $p2 - p3$

5、指在同一个数组里的指针可以比较大小（前后）：

$\text{if} (p3 > p2) \dots$

6、两个同类型指针可用  $==$  和  $!=$  比较相等或不等；任何指针都能与通用指针比较相等或不等，任何指针可与空指针值（0 或 NULL）比较相等或不等。

两指针指向同一数据元素，或同为空值时它们相等。

## 指针运算小结

一个指针指在数组里，可以：

- 普通加减法，然后做间接访问  $*(p1+2)$   $*(p2-1)$
- 自增、自减  $p1++$ ;  $p2--$

两个指针指在同一数组里，可以：

- 两个指针相减，求相差个数  $p1-p2$
- 两个指针比较大小，求前后关系  $p1 > p2$

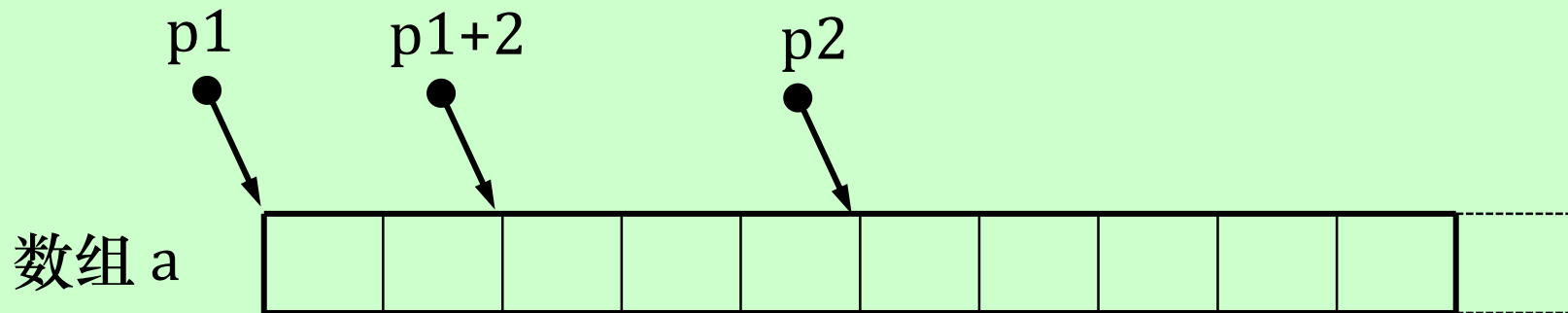
同一类型的两个指针，可以：比较是否相等

## 指针运算原理

当一个指针指向某数组里的元素时，为什么能计算出下一元素位置？（这是指针运算的基础）

**指针有指向类型**，p 指向数组 a 时，由于 p 的指向类型与 a 的元素类型一致，数据对象的大小（存储空间字节数）可以确定。

p+1 的值可根据 **p 的值** 和 **数组元素大小** 算出。由一个数组元素位置可以算出下一元素位置，或几个元素之后的元素位置。



通用指针即使指到数组里，因没有确定指向类型，因此不能做一般指针计算，只能做指针比较。

## 7.3.2 数组写法与指针写法

如果一个指针指在一个数组里，通过指针访问数组元素的操作也可用下标形式书写。可以把该指针所指位置视为一个数组的首元素。



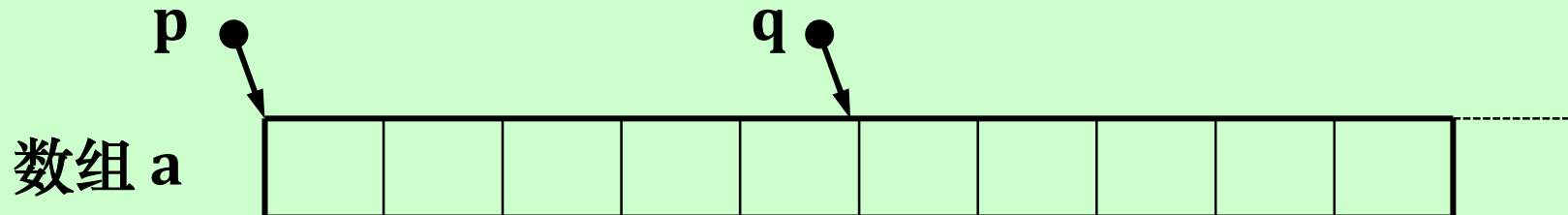
设  $p$  指向数组  $a[0]$ ， $q$  指向  $a[5]$ 。可写：

$p[0] = 4$ ;  $p[3] = 5$ ;  $q[0] = 6$ ;  $q[2] = 8$ ;

$p[3]$  和  $q[2]$  这类写法称为数组写法，

前面说的  $*(p+3)$  和  $*(q+2)$  这类写法称为指针写法。

两类写法有等价效力，可以自由选用。

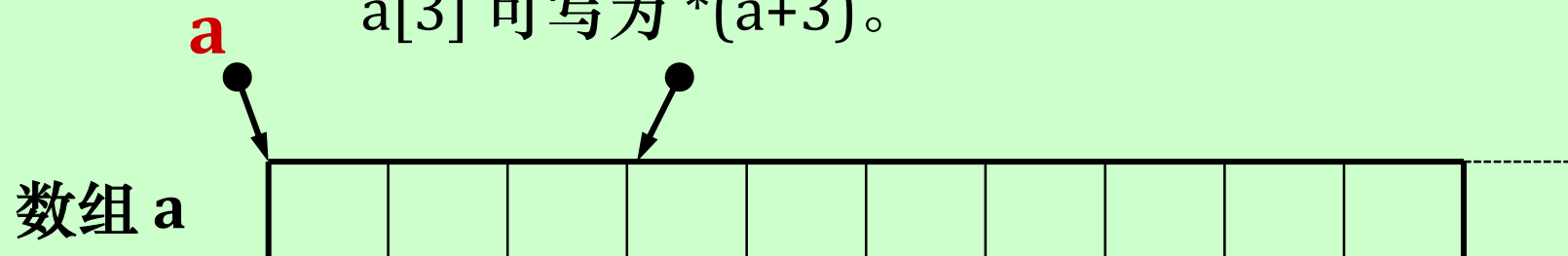


对数组名求值  $*a$  得到指向数组首元素的指针值。

数组名可以“看作”常量指针，可参与一些指针运算，与其他指针比大小，比较相等与不相等。

通过数组名的元素访问也可以采用指针写法。

$a[3]$  可写为  $*(a+3)$ 。



注意：数组名不是指针变量，特别是不能赋值，不能更改。  
下面操作都是错误的：

$a++;$   $a += 3;$   $a = p;$

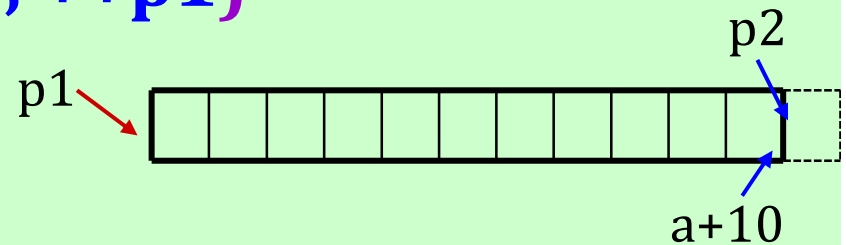
有些运算虽不赋值但也可能没意义。如  $a-3$  不可能得到合法指针值，因其结果超出数组界限。

## 7.3.3 基于指针的数组程序设计

指针运算是处理数组元素的另一方式，有时很方便。

设有 int 数组 a 和指针 p1, p2，打印 a 的元素：

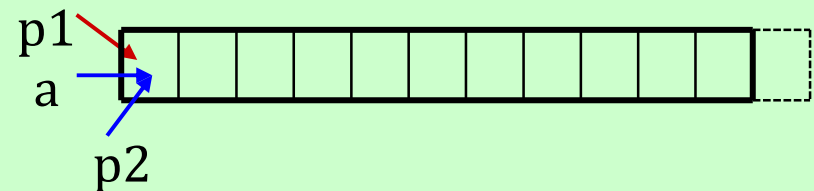
```
for (p1 = a, p2 = a+10; p1 < p2; ++p1)
    cout << *p1 << endl;
```



```
for (p1 = a; p1 < a+10; ++p1)
    cout << *p1 << endl;
```

```
for (p1 = p2 = a; p1 - p2 < 10; ++p1)
    cout << *p1 << endl;
```

```
for (p1 = a; p1 - a < 10; ++p1)
    cout << *p1 << endl;
```



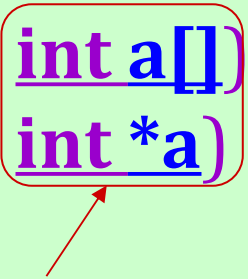


## 7.3.4 数组参数与指针

前面讲到：在执行有数组参数的函数时，对形参数组的元素操作就是直接对函数调用时的实参数组的元素操作。为什么？解释如下：

函数的数组参数就是相应的指针参数。例如：

`int func(int n, int a[]) {...}` 与  
`int func(int n, int *a) {...}` 意义相同。



实参中的数组名传递给形参的是指向该数组“首元素”的指针。函数内部通过指针间接访问相应实参数组里的各元素。从而可以修改实参数组。

```
int func(int n, int a[]) {...}  
int func(int n, int *a) {...}
```

这是指针！

函数里不能用 `sizeof` 确定数组实参大小：函数的数组形参实际是指针，求 `sizeof` 算出的是指针的大小。

所有指针大小都一样，它们保存的都是地址值，各种类型的地址值采用同样表示方式。

另一方面，`sizeof` 的计算是在编译中完成的。实参是动态运行中确定的东西。

因为数组参数实际上传递的是指针，所以含有数组参数的函数可以有多种灵活的用法。

假设有函数可求元素之和：

```
double sum(int n, double a[]);
```

程序中有双精度数组 b: `double b[40];`

则可以用多种方法灵活调用该函数：

```
x = sum(40, b);      //处理整个数组 b
```

```
y = sum(20, b);      //只处理前20个元素
```

```
double *pb=&b[17]; //指针指向数组
```

```
y = sum(11, pb);     //处理从下标17开始的11个元素
```

```
y= sum (11, b+17);
```

## 7.3.5 多维数组作为参数的通用函数

函数的两维或多维数组参数必须说明除第一维外各维的大小，这使函数失去了一般通用性。

ANSI C 没提供定义处理多维数组的通用函数的标准方法。可以通过技术解决。（C99 提供了标准方式）

下面以两维数组为例，多维数组可以类似处理。

考虑：

```
int fun1(int n, int mat[][10])  
{ ... .. mat[i][j] ... .. }
```

只能对第二维长10的数组使用。不能处理其他数组。

改写为：

```
int fun2(int n, int m, int mat[][])  
{ ... .. mat[i][j] ... .. }
```

这个定义错误，无法通过编译。为什么？

mat 的指向类型是 int 数组（二维数组的元素是一维数组）。定义没给出一维数组大小，指针定义不完全。

这样，编译程序虽然知道 mat[0] 的位置，但却无法算出 mat[1] 等子数组位置以及 mat[i][j] 位置。因此编译工作无法完成。

实际中确实需定义处理多维数组的通用函数。

## 解决方案：以指针写法访问数据元素

考虑数组 `int a[10][8];`

首元位置 `&a[0][0]`。

每行 8 元素，第1行开始位置是：

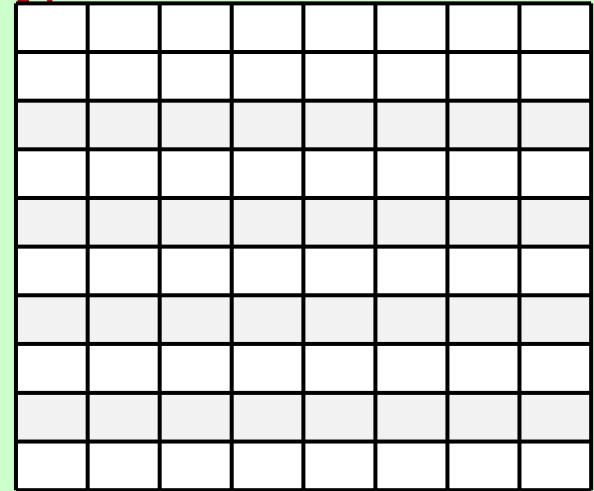
$$\&a[0][0] + 8$$

访问第 1 行首元素用表达式： `*(&a[0][0] + 8)`

访问第  $i$  行首元素用表达式： `*(&a[0][0] + i*8)`

访问 `a[i][j]`： `*(&a[0][0] + i*8 + j)`

可见，有了一些信息后，可以算出元素的位置。



处理二维数组所需信息：（1）基本元素类型；（2）数组两个维的长度；（3）数组开始位置。

通过参数可得到数组的首元素位置和各维长。

**【例】** 输出二维整型数组的函数，每行输出在一行。

```
void prtMatrix (int m, int n, int *mp){  
    int i, j;  
    for (i = 0; i < m; ++i) {  
        for (j = 0; j < n; ++j)  
            cout << *(mp + i * n + j) << "\t";  
        putchar('\n');  
    }  
}
```

打印数组a和另一 $20 \times 36$ 的整型数组mat的调用形式：

```
prtMatrix(10, 8, &a[0][0]);  
prtMatrix(20, 36, &mat[0][0]);
```

## 7.3.6 指针与数组操作函数实例

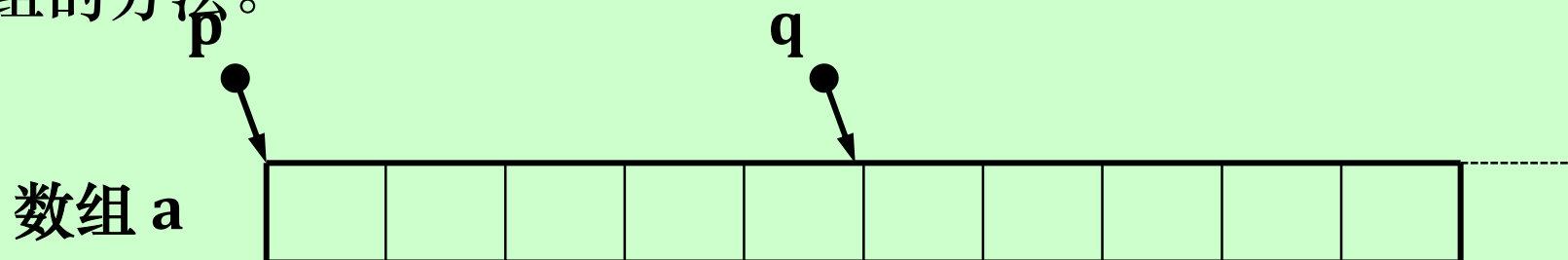
字符指针 (char\*) 应是指向字符变量。

```
char ch='a', *pch = &ch;
```

但在实际应用中，由于一个字符变量仅能存储一个字符，所发挥的作用及其有限，所以这种用法并不常见。

人们常常用字符指针指向字符数组的元素，以便通过这种指针来操作字符数组的元素。

本节里用几个完成字符串操作的函数实例说明通过指针操作数组的方法。

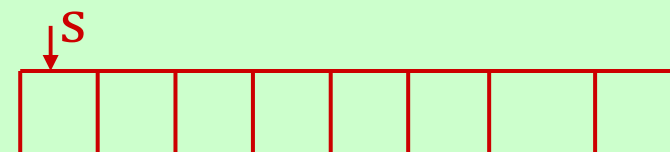




## 【例7-3】 用指针方式实现字符串长度函数。

方式一：

```
int strLength (const char *s) {  
    int n = 0; //通过局部指针扫描串中字符并计数  
    while (*s != '\0') { s++; n++; }  
    return n;  
}
```



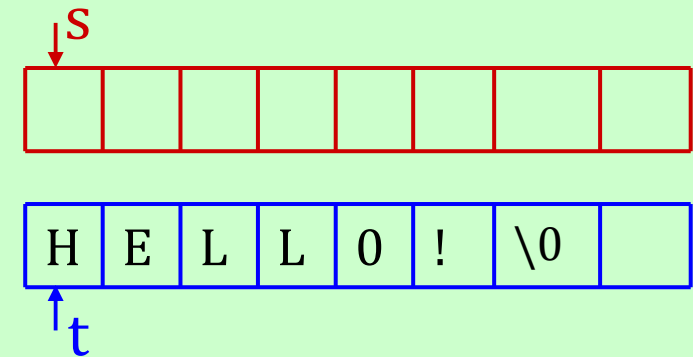
参数类型(char\*), 实参应该是  
常量字符串或存字符串的数组

另一实现：

```
int strLength (const char *s) {  
    const char *p = s;  
    while (*p != '\0') p++;  
    return p - s; //通过局部指针扫描到串尾，计算指针之差  
}
```

## 【例7-4】用指针实现字符串复制函数。直接定义：

```
void strCopy (char *s, const char *t) {  
    while ((*s = *t) != '\0') {  
        s++; t++;  
    }  
}
```

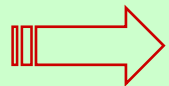


赋值表达式有值，'\0'就是0，函数可简化：

```
void strCopy (char *s, const char *t) {  
    while (*s = *t) { s++; t++; }  
}
```

把指针更新操作也写在循环测试条件里：

```
void strCopy (char *s, const char *t) {  
    while (*s++ = *t++); // 循环体为空语句  
}
```



注意优先级与结合性，增量运算的作用与值等。

表达式 \*s++ = \*t++ 怎么理解？

\* 运算符与 ++ 运算符的优先级相同，结合性是从右向左。

据此分析，该表达式的语义是： \*(s++) = \*(t++)

通俗地说（容易理解但含有知识性错误），可以这样理解：

后置自增运算符是先做别的运算，然后再做自增；

所以该式等价于： \*s=\*t; s++; t++;

准确地说（知识准确但不易理解），先对两个自增表达式 (s++) 和 (t++) 求值，则 s 和 t 的值增加了；同时这两个表达式分别返回 s 和 t 自增之前的值（假设以 s0 和 t0 表示），然后再作间接访问和赋值： \*s0 = \*t0。

【例7-5】 利用指针，输出 int 数组里一段元素：

```
void prt_seq(int *begin, int *end) { // 区间 “左闭右开”  
    for (; begin != end; ++begin)  
        cout << *begin << endl;  
}
```

```
prt_seq(a, a+10);  
prt_seq(a+5, a+10);  
prt_seq(a, a+3);  
prt_seq(a+2, a+6);  
prt_seq(a+4, a+4);  
prt_seq(a+10, a+10);
```

最后两个调用对应空序列。

还可写出许多类似函数。“设置”函数：

```
void set_seq(int *b, int *e, int v)
{   for (; b != e; ++b) *b = v; }
```

把序列中每个元素都用其平方根取代：

```
void sqrt_seq (double *b, double *e)
{   for (; b!=e; ++b) *b = sqrt(*b); }
```

求平均值：

```
double avrg(double *b, double *e) {
    double *p, x = 0.0;
    if (b == e) return 0.0;
    for (p = b; p != e; ++p) x += *p;
    return x / (e - b);
}
```

## 小结

常用字符指针指向字符数组的元素，以便通过这种指针来操作字符数组的元素。

【例7-3】用指针方式实现字符串长度函数

- 1) 通过局部指针扫描串中字符并计数
- 2) 通过局部指针扫描到串尾，计算指针之差

【例7-4】用指针实现字符串复制函数

在循环中做间接访问赋值和指针更新。可以有多种写法。尤其是 `*s++ = *t++`，需要仔细体会其含义。

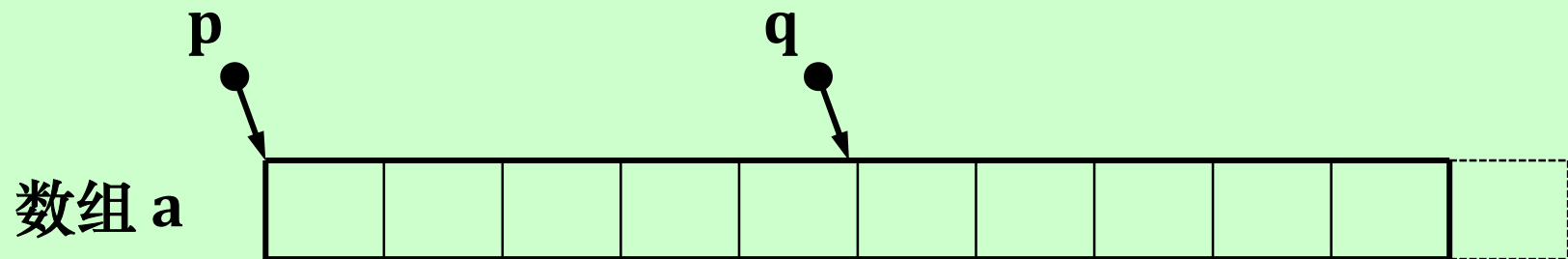
【例7-5】利用指针，输出 int 数组里一段元素  
输出两个指针之间“左闭右开”区间中的数组元素。

## 7.3.7 字符指针与字符数组

char \*p

char str[10]

常用字符指针指向字符数组元素。如指向常量字符串或存着字符串的字符数组，通常指向字符串开始。



字符指针还有另一种常见用法，即**让字符指针指向字符串**：或是**指向一个常量字符串**，或是指向存储着字符串的字符数组。

通常情况是字符指针指向字符串的开始，然后使用该字符指针去操作该字符串。

补充说明：“字符串(string)”相关知识。

通俗地说，“字符串”就是把一些字符组合成一个串（词组，段落，甚至文章），作为一个整体来处理。

C 语言中没有“字符串(string)”这个类型(type)。通常用两种方式处理字符串：

(1) 用户直接拿一个常量字符串给系统，系统自动安排内存去处理。通常是一次性使用的。例如：

```
cout << "Hello, world!\n";
```

(2) C 语言里通常用字符数组存储字符串（至少要存储一个'\0'用于表示字符串结束）。在标准库中提供了一些对字符串操作的函数。用户可以定义一个字符数组，然后用各种字符串函数去处理这个数组。例如：

```
char str[20];          strcpy (str, "Hello, World!\n");
```

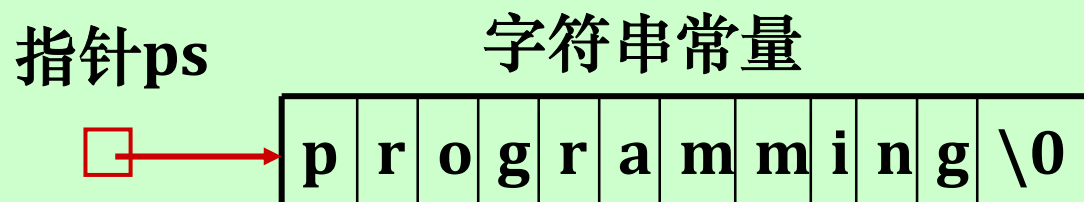
（在 C++ 语言中，有“string”和“CString”这两个类(Class)专门用于处理字符串。）



定义字符指针时可用常量字符串初始化。

```
char *ps = "Programming";
```

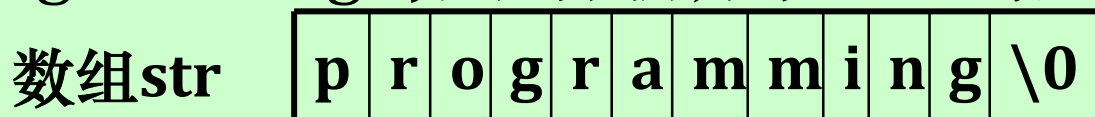
- 1) 定义了指针 ps
- 2) 建立了一个字符串常量，内容为 "Programming"
- 3) 令 ps 指向该字符串常量。

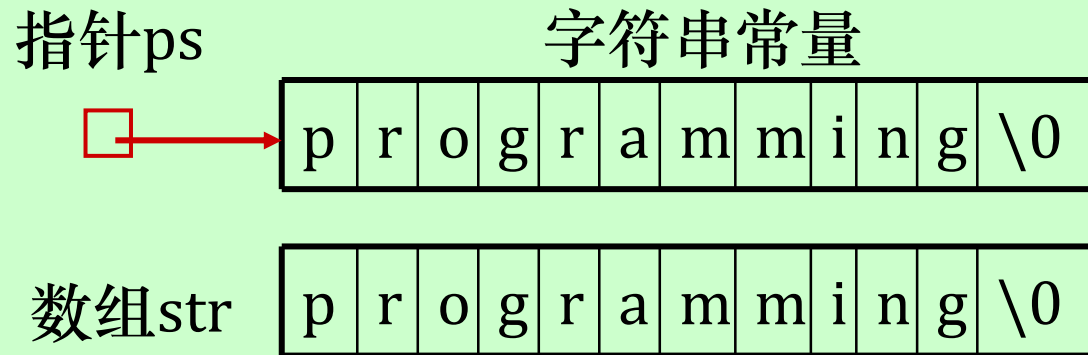


定义字符数组时也可用常量字符串初始化。

```
char str[] = "Programming";
```

- 1) 定义了一个12个字符元素的数组str
- 2) 用 "Programming" 各字符初始化 str 的元素





1) 指针 ps 可重新赋值（数组不能赋值）：

ps = "Programming Language C";

2) ps 和 str 类型不同，大小不同。

3) str 的元素可以重新赋值。如：

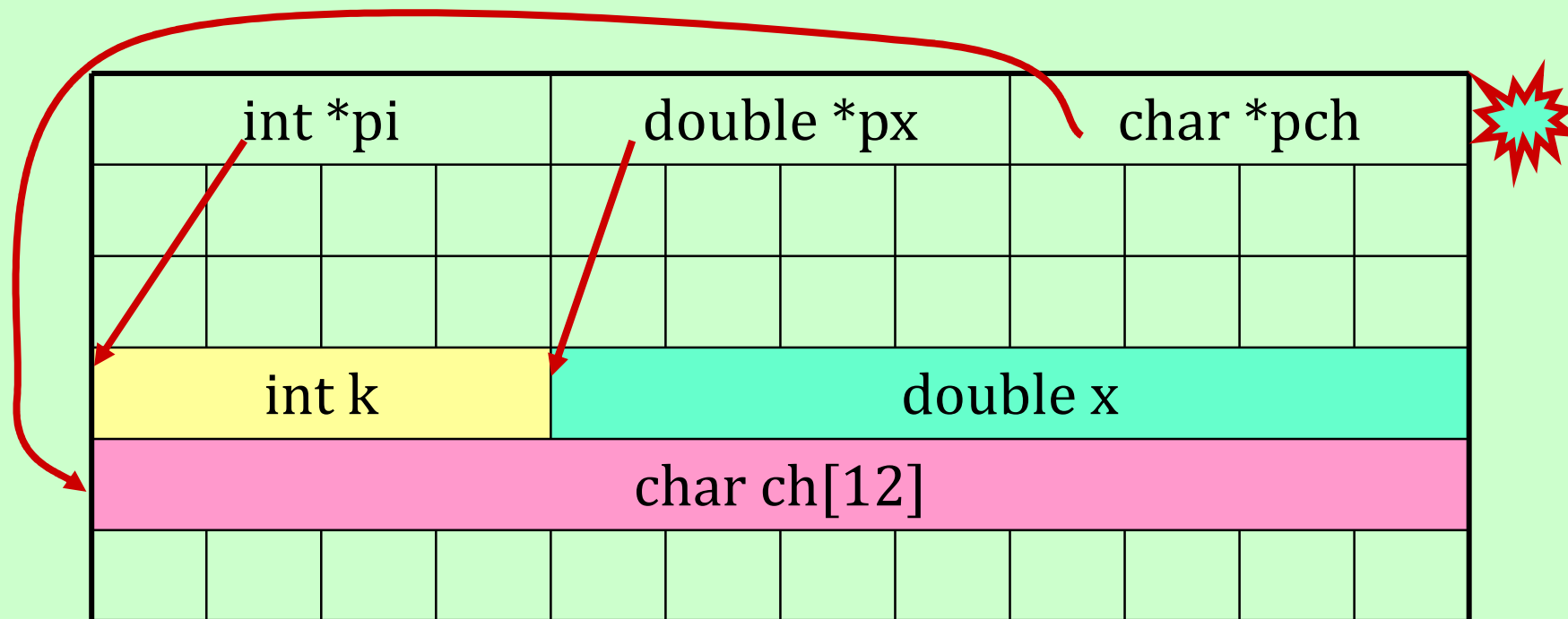
str[8]='e'; str[9]='r'; str[10]='\0';

str 的内容现在变成 “Programmer”

按规定，字符串常量不得修改。

## 补充知识

问：每个变量在程序运行时都会占用内存空间，那么**指针变量**在内存占多少位？与类型有关吗？



答：与类型无类，取决于编译器。经由32位编译器编译出来，都只占32位(4个字节)。经由64位编译器编译出来，都占64位(8个字节)。

## 基础知识

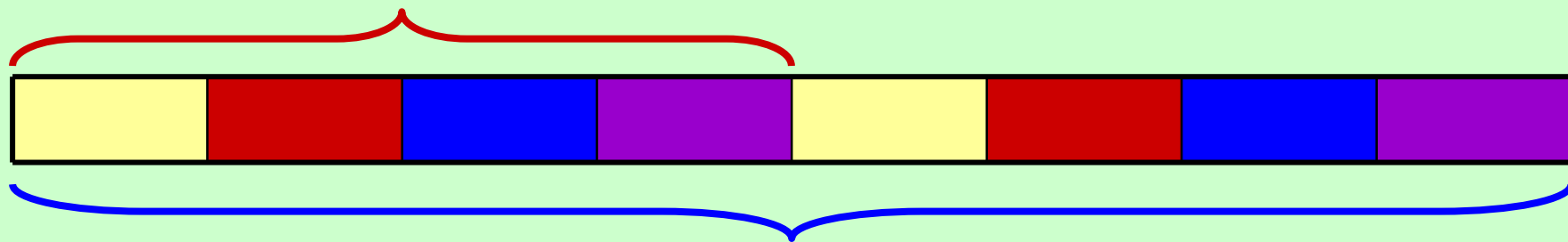
问：“32位”和“64位”是什么意思？

**字(Word)**：是计算机系统（硬件和软件）进行数据存储传送和处理的数据信息运算单位，是由若干字节组成。每个字所占的位数称之为**字长**。

字长越长，处理速度就越快。



32位系统在单位时间内能处理字长为 32 位（4字节）的二进制数据

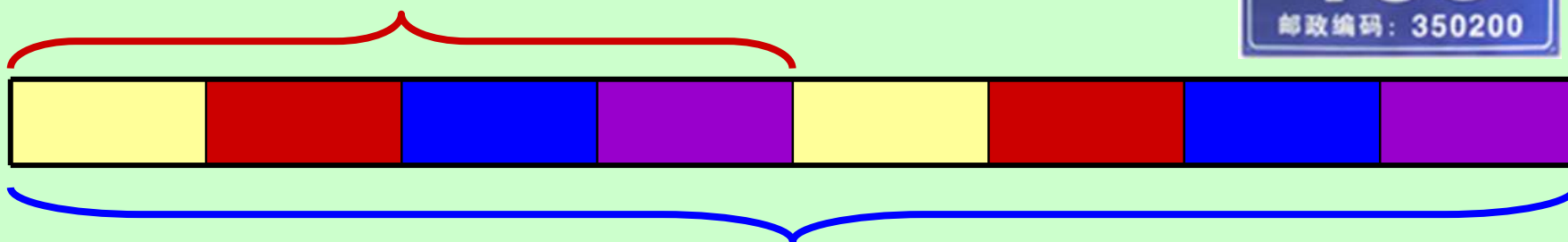


64位系统在单位时间内能处理字长为 64 位（8字节）的二进制数据

## 基础知识

- 系统按照字长给每个内存单位指定地址（“编号”）。每个地址必须唯一、不重复。
- 因此地址编号取决于字长，数量也取决于字长。

32位系统用 32 位（4字节）存储地址



- 指针变量存储的是内存地址，长度即为字长。
- 编译器的字长是根据系统而定的。

## 试一试

//用 sizeof 函数计算变量所占内存长度

```
# include<stdio.h>
int main() {
    int i = 373, *p = &i;
    double x = 4.5, *q = &x;
    char ch = 'A', *r = &ch;
    printf("%d %d %d\n",sizeof(i), sizeof(x), sizeof(ch));
    printf("%d %d %d\n",sizeof(p), sizeof(q), sizeof(r));
    return 0;
}
```

32位: 4 8 1 4 4 4

64位: 4 8 1 8 8 8

- 进一步的问题：指针的类型信息存放在哪里？  
比如有语句 `char* p1;` 系统怎么知道 `p1` 的类型是 `char *`，而不是 `int*` 呢？
- C 语言的指针类型包括两方面的信息：一是地址，存放在指针变量中；二是类型信息，关系到读写的长度。类型信息并没有存储在指针变量中，而是在编译时由编译器确定的，体现在编译器生成的汇编指令中。对于不同类型的指针，编译后生成的汇编指令 “`mov`” 不同。
- “类型” 概念只存在于编译前，编译器会识别出各种数据类型，然后生成目标文件，`obj` 文件里面就包含了每条语句的汇编实现。在运行期不需要关心 “类型”。